Design and Development of a QRCode-Based Autonomous Navigation System for Turtlebot4

Andrea Vincenzo Ricciardi [2009], **Giovanni Rolando** [2018], **Andrea Zinno** [2064], **Ciro Ciaravolo** [2006]

[1]Dept. of Information Eng., Electrical Eng. and Applied Mathematics - University of Salerno, Italy

Abstract The main objective of this project is to design and develop a software system that enables the Turtlebot4 to autonomously navigate within a predefined environment at the Department of Industrial Engineering and Management (DIEM). The project focuses on leveraging a priori knowledge of the environment's map to facilitate the Turtlebot4's movement from any given starting position, ensuring it can discover and follow a path during navigation by interpreting commands provided through QR codes. This initiative employs a command-based navigation strategy where QR codes placed at various intersections within the environment serve as navigation beacons. These QR codes encode specific commands that direct the Turtlebot4 to perform actions such as turning right, turning left, moving straight, or stopping. The Turtlebot4's ability to autonomously read and interpret these commands is crucial for its successful navigation. The robot must traverse corridors, approach intersections, read the QR codes, and make informed decisions to follow the encoded instructions. This system is designed to handle an indeterminate number of obstacles that might be placed along the robot's path, ensuring robustness and adaptability in dynamic environments.

For correspondence:

a.ricciardi38@studenti.unisa.it (AVR); a.zinno6@studenti.unisa.it (AZ); g.rolando1@studenti.unisa.it (GR); c.ciaravolo@studenti.unisa.it (CC)

1 | Introduction

The focus of this project is to enable autonomous navigation of the Turtlebot4 within a controlled environment, hereafter referred to as `@DIEM`. Leveraging the capabilities of ROS2, specifically the Navigation Stack 2, the project aims to facilitate seamless movement of the Turtlebot4 through predefined paths using QR code commands. These commands direct the robot to perform manoeuvrers such as turning, moving straight, and stopping at intersections marked by QR codes. The Navigation Stack 2 in ROS2 serves as the backbone of this autonomous navigation system, integrating localization, path planning, and execution functionalities to ensure precise and efficient movement within the `@DIEM` environment.

The following sections detail the methodology employed, including the design choices, technical implementations such as waypoint representation using graph structures, and the integration of QR code detection for command execution. Furthermore, this paper comprehensively explores the performance evaluation of the implemented navigation system through a diverse series of tests. These tests aim to rigorously assess the system's functionality across various scenarios and conditions. Each test scenario is designed to simulate realistic challenges that the Turtlebot4 might encounter during autonomous navigation tasks within the `@DIEM` environment.

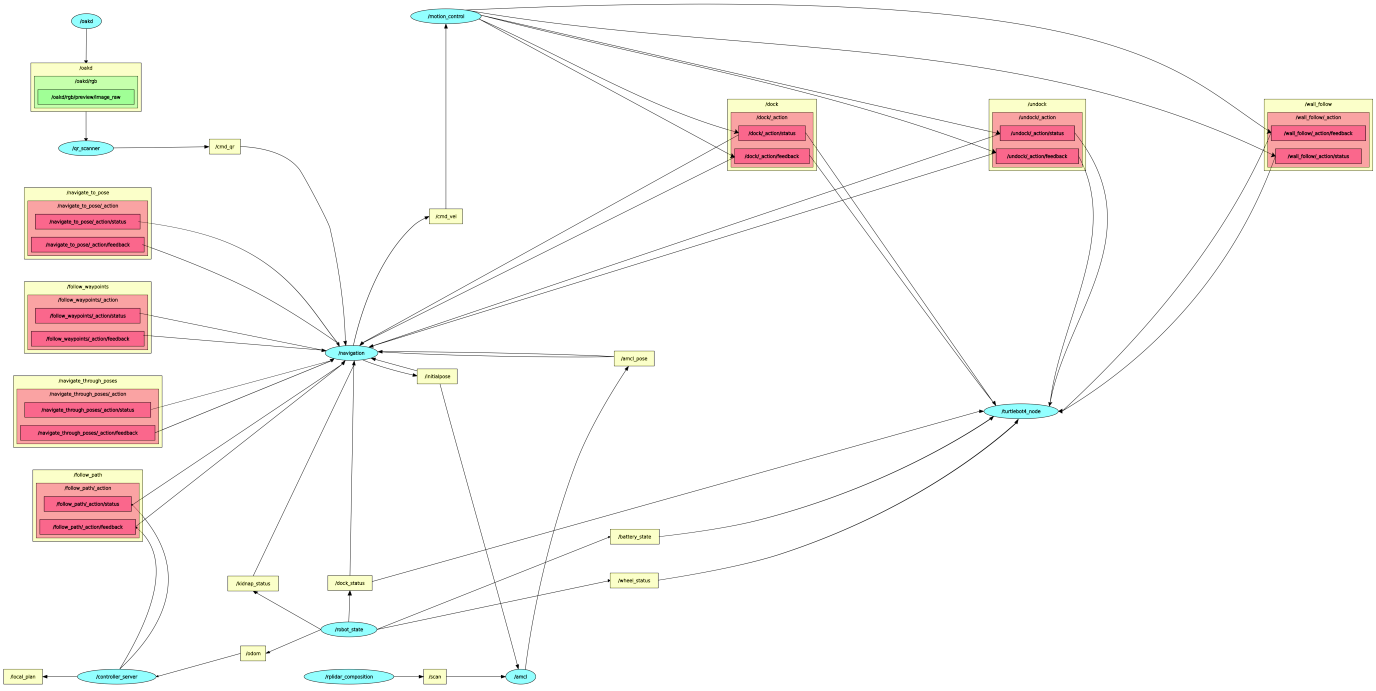


Fig. 1 – Comprehensive high-level architecture of the implemented navigation system. Legend: `yellow` indicates ROS2 topics such used for communication between nodes; `blue` represents ROS2 nodes executing computations; `red` denotes ROS2 actions offering mechanisms for time-consuming tasks with feedback.

2 | High-level Architecture

As depicted in Fig. 1 and previously introduced, our navigation system, implemented in the `navigation` node, is heavily based on the **NAV2** navigation framework. Its primary goal is to enable mobile robots to navigate reliably and safely through complex environments to reach user-specified goals. In our case, this involves traversing `@Diem` map by utilizing information from the Turtlebot4's sensors.

Nav2 As the backbone of our navigation system, **Nav2** facilitates the seamless integration of various navigation components, ensuring robust path planning and execution. It offers a variety of actions and services that are crucial for enabling navigation between different waypoints on the map. One of the key actions we utilize is the `NavigateToPose` action. This action is used to navigate the robot to a target pose, which is determined by reading the QR code placed at specific locations. By leveraging the feedback provided by this action, we can implement specific logic in our code to facilitate the generation of expected behaviours from the Turtlebot4. This includes updates on whether the robot is actively en route to its destination, has successfully reached the designated target, or if the navigation goal has been unexpectedly cancelled. In addition to actions, NAV2 provides several services that enhance the functionality and reliability of the navigation system. One such service is the `clear_all_costmaps` service, which we utilize to clear the costmaps at the end of each completed segment of the route.

QRScanner Turtlebot4 utilizes data from various sensors to enhance its navigation capabilities. Camera data plays a significant role in augmenting the navigation process. Specifically, the camera is employed to detect QR codes strategically positioned throughout the `@Diem` map. The `qr_scanner` node within our system publishes decoded commands onto the `/cmd_qr` topic derived from reading the QR codes. If a QR code is successfully detected and deciphered, the node publishes the corresponding command the topic. Conversely, if no QR code is detected or if the message cannot be decrypted, the node abstains from publishing any commands.

3 | Design Choice Implemented

In this section, we delve into the methodology adopted to enhance the autonomous navigation capabilities of the Turtlebot4 within the `@DIEM` environment. Our project aims to develop sophisticated software that enables the Turtlebot4 to autonomously navigate within a predefined environment.

3.1 | Waypoints Representation

The navigation of Turtlebot4 via commands provided through QR codes necessitates a structure that can easily manage the transition between states and the execution of commands that alter the path. A **graph** structure provides a clear and intuitive way to represent the environment, with vertices (nodes) representing significant locations and edges (links) the traversable paths between these locations. This is particularly suitable for indoor environments where paths are well-defined, and the connections between points are crucial for navigation. Each node can represent a waypoint, and edges denote possible routes the Turtlebot4 can take, making it straightforward to visualize and manage the spatial layout. To obtain the positions of waypoints, we utilized RViz to determine their coordinates relative to the map. It allows us to accurately pinpoint the locations of waypoints within the environment and establish their coordinates with respect to a reference frame, such as the `@DIEM` map. Once the coordinates of each waypoint were determined, we recorded this information in a JSON file for seamless integration with our navigation system.

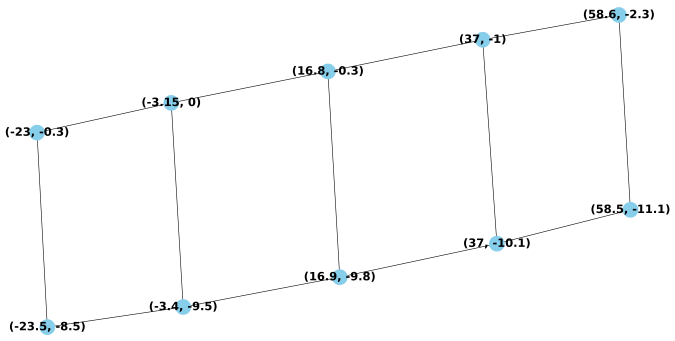


Fig. 2 – Representation of the graph whose vertices correspond to the waypoints positioned on the `@Diem` map.

```
{
    "1": {
        "source": {
            "x": -23,
            "y": -0.3
        },
        "target": {
            "x": -3.15,
            "y": 0
        }
    },
    "2": {}
}
```

Listing 1 – Structure of the JSON file for storing the possible edges of the graph. It adheres to a structured format where each edge, representing a traversable path for the Turtlebot4, is uniquely identified by an identifier (e.g., "1", "2", etc.). Each entry in the JSON file comprises two points: the source and the target waypoint.

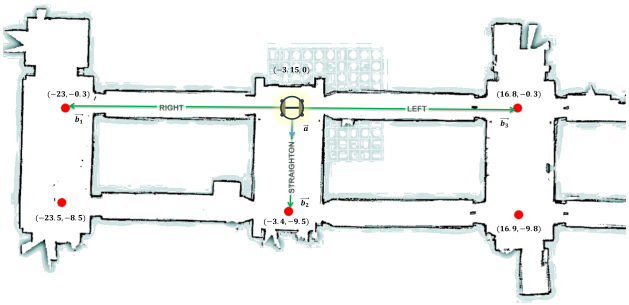
3.2 | Get the next waypoint

The process of determining the next waypoint for the Turtlebot4 to follow is a critical aspect of its autonomous navigation system. This calculation involves intricate geometric computations based on the robot's current orientation and the potential movement directions within the environment. Before calculating the next waypoint, it's essential to establish the orientation vector of the Turtlebot4. This vector represents the direction in which the robot is currently facing and is typically derived from its pose information. The orientation vector, denoted as **a**, serves as a reference direction against which the potential movement directions will be evaluated. It is computed using the cosine and sine components of the yaw angle, which is the angle of rotation around the vertical axis:

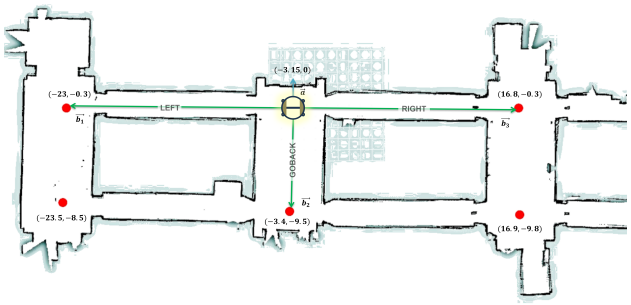
$$\mathbf{a} = \begin{bmatrix} \cos(\text{yaw}) & \sin(\text{yaw}) \end{bmatrix}^T \quad (1)$$

To identify the next waypoint, the system considers the edges adjacent to the Turtlebot4's current position. For each edge, a displacement vector, denoted as **b**, is computed. By subtracting the coordinates of the current waypoint from those of the candidate waypoint, the components of **b** are determined, encapsulating the potential movements the Turtlebot4 can make:

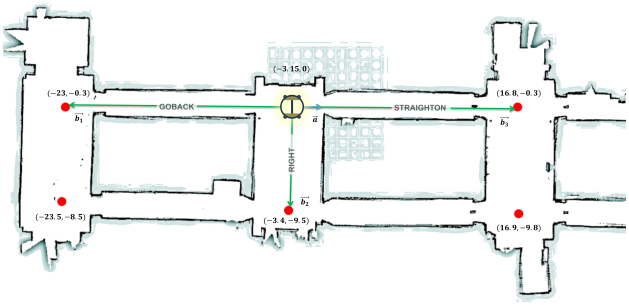
$$\mathbf{b} = \begin{bmatrix} \text{next\_wp.x} - \text{curr\_wp.x} & \text{next\_wp.y} - \text{curr\_wp.y} \end{bmatrix}^T \quad (2)$$



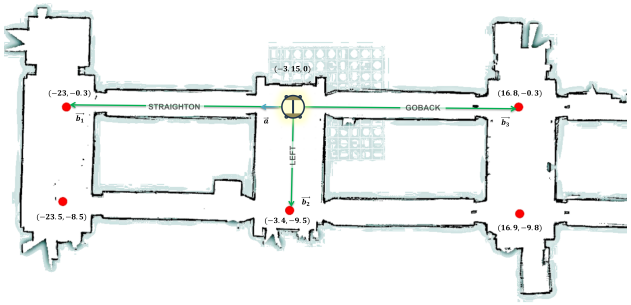
(a) Turtlebot4 approaching an intersection from the North.



(b) Turtlebot4 approaching an intersection from the South.



(c) Turtlebot4 approaching an intersection from the East.



(d) Turtlebot4 approaching an intersection from the West.

Fig. 3 – Illustrative scenarios depicting Turtlebot4 approaching from various directions and potential orientations.

To understand the direction in which the Turtlebot4 should proceed, the system utilizes two geometric operations:

- **Cross Product $\mathbf{a} \times \mathbf{b}$.** It provides insights into the relative orientation between the orientation vector **a** and the displacement vector **b**. By examining the sign of the cross product, the system determines whether the next waypoint is to the left or right of the Turtlebot4's current orientation. A positive cross product indicates a leftward orientation, while a negative cross product indicates a rightward orientation. The general formula for the cross product in Cartesian components is:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \quad 3$$

  Since we are working in a two-dimensional space, the resulting vector has only a *k*-component, which represents the perpendicular direction to the plane. Therefore, the cross product simplifies to:

$$\mathbf{a} \times \mathbf{b} = \left(a_x \cdot b_y - a_y \cdot b_x\right) \quad 4$$

- **Dot Product $\mathbf{a} \cdot \mathbf{b}$.** It quantifies the alignment between the orientation vector **a** and the displacement vector **b**. This calculation reveals whether the next waypoint lies ahead or behind the Turtlebot4's current orientation. A positive dot product indicates alignment in the same direction, suggesting that the next waypoint is ahead, while a negative dot product indicates alignment in the opposite direction, suggesting that the next waypoint is behind the Turtlebot4. Since we are working in a two-dimensional space, the dot product simplifies to:

$$\mathbf{a} \cdot \mathbf{b} = \left(a_x \cdot b_x + a_y \cdot b_y\right) \quad 5$$

Once the desired direction is determined, the system selects the appropriate waypoint from the neighboring positions of the current waypoint. This ensures that the Turtlebot4 adheres to the correct path according to the instructions provided by the QRCode commands.

3.3 | QR Detector

The QR Detector module within our project serves as a pivotal component, enabling our Turtlebot4 to interpret QR codes found on signs and make informed decisions based on the information contained within them. At the heart of this functionality lies the open-source tool called `QReader` [1]. It is a robust and straightforward solution for reading complex and problematic QR codes within images, implemented in Python. The library is composed by two main building blocks:

- **YOLOv8 QR Detector**: The chosen and pre-trained model for detecting and segmenting QR codes. This model is included in `QReader` and can also be used separately.
- **Pyzbar QR Detector**: `QReader` uses the information extracted by this QR detector to apply image preprocessing techniques that maximize the decoding rate on challenging images.

Using the information extracted from this QR Detector, `QReader` transparently applies, on top of `Pyzbar`, different image preprocessing techniques that maximize the decoding rate on difficult images. This module enables our Turtlebot4 to interpret QR codes efficiently, allowing it to respond dynamically to various commands encoded within them, such as `STOP`, `LEFT`, `RIGHT`, `STRAIGHTON`, and `GOBACK`. With `QReader`, the Turtlebot4 can navigate its environment intelligently, making informed decisions based on the information provided by QR codes scattered throughout the environment.

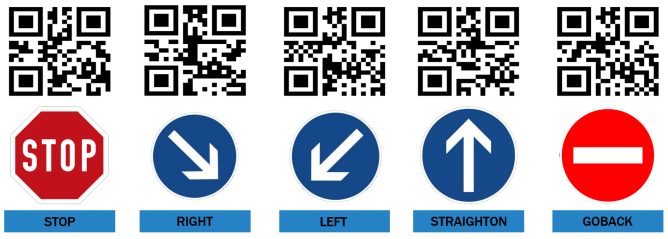


Fig. 4 – Set of commands that we had to take into account.

At the core of the QReader library lies its primary class, which encapsulates the essential functionalities for QR code detection and decoding. The instantiation of this class initializes crucial parameters and resources necessary for subsequent operations. The constructor method, denoted as QReader, offers flexibility in configuring the model size, minimum confidence threshold, and encoding for reencoding decoded strings. Specifically:

- `model_size` (str): Determines the size of the model utilized for QR code detection, ranging from `'n'` (nano) to `'l'` (large). While larger models yield heightened accuracy, they incur higher computational overhead. Defaulting to `'s'` (small), this parameter allows a trade-off between accuracy and efficiency. To ensure that our Turtlebot4 remains nimble and responsive during its navigation tasks, we've opted for the `'n'` model size. This choice prioritizes computational efficiency without compromising significantly on detection accuracy.
- `min_confidence` (float): Establishes the minimum confidence threshold for a QR detection to be deemed valid. A delicate balance exists between minimizing false positives and capturing challenging QR codes, with a default and recommended value of 0.5.
- `reencode_to` (str | None): Offers the flexibility to specify an encoding for reencoding the UTF-8 decoded QR strings. This parameter serves as a mechanism to rectify inaccurately decoded characters, with recommended encodings including `'shift-jis'` for Germanic languages and `'cp65001'` for Asian languages.

The primary methods within the QR Detector module are meticulously crafted to deliver swift and accurate QR code detection and decoding. Leveraging the capabilities of the `'n'` model, the `detect_and_decode` method swiftly processes input images, extracting pertinent QR code information with minimal latency. Additionally, the `detect` method provides detailed detection information for each discerned QR code, empowering our Turtlebot4 with rich contextual insights for informed decision-making.

3.4 | Navigation Manager Logic

The **Navigation Manager** is a critical component responsible for guiding the robot through a series of waypoints on a predefined map. This component ensures that the robot moves efficiently and accurately to its target destinations while responding to commands received from QR codes. The navigation process involves several key steps, each designed to maintain the robot's trajectory and handle any unexpected interruptions.

Initialization and Pose Setting The navigation process begins with the initialization of the robot's position and the waiting period for the navigation stack to be ready. This ensures that all necessary components, such as the localization system (AMCL) and the navigation stack, are fully operational before the robot starts moving. The initial pose of the robot is set based on the current position received from the `/initial_pose` topic, which is only done once at the beginning of the navigation to establish a starting reference point.

Navigation Loop Once the initial setup is complete, the robot enters a continuous navigation loop where it processes commands and moves through a series of waypoints. The robot's movement is governed by commands decoded from QR codes, with a default behaviour to move straight ahead (`Commands.STRAIGHTON`) until a new command is received. This loop ensures that the robot constantly updates its position and follows the designated path on the map.

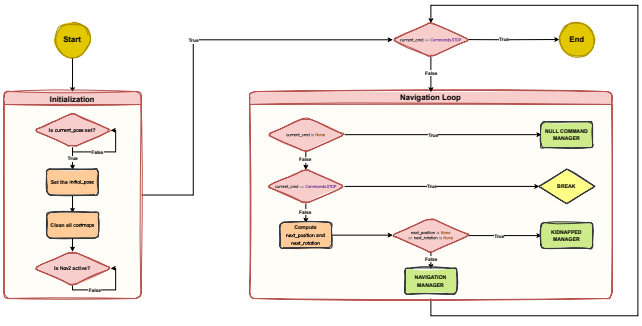


Fig. 5 – Flowchart illustrating the operation of the `loop` method in the Turtlebot4 navigation system.

Command Processing and Waypoint Navigation In each iteration of the loop, the robot checks for new commands and determines the nearest waypoint from its current position. This is achieved by calculating the Euclidean distances between the robot's current location and the predefined waypoints on the map. The robot then calculates the next position and rotation required to reach the subsequent waypoint, based on the current command.

Movement Execution The Navigation Manager moves the robot to the next waypoint using the following steps:

- **Goal Pose Creation:** A goal pose is created based on the next position and rotation.
- **Path Planning and Execution:** The robot sends the goal pose to the navigator, which plans and executes the path to the target waypoint. The robot adjusts its speed and orientation to reach the goal efficiently.
- **Orientation Adjustment:** Upon reaching the waypoint, the robot adjusts its orientation to align precisely with the target rotation. This step is critical to ensure the robot's direction of movement matches the intended path. Proper alignment is essential because any deviation could render the entire navigation system ineffective and compromise the robot's ability to follow subsequent waypoints accurately. This step involves calculating the angular difference between the current orientation and the target orientation:

$$\delta_{\text{rot}} = \bar{\theta}_{\text{target}} - \bar{\theta}_{\text{current}} \tag{6}$$

  where $\bar{\theta}_{\text{target}}$ is the target orientation and $\bar{\theta}_{\text{current}}$ is the current orientation of the robot.
- **Costmap Clearing:** The costmaps of the local and global planners are cleared to remove any previously detected obstacles and replan the path if necessary.

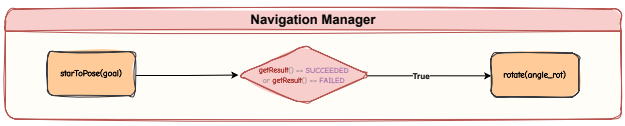
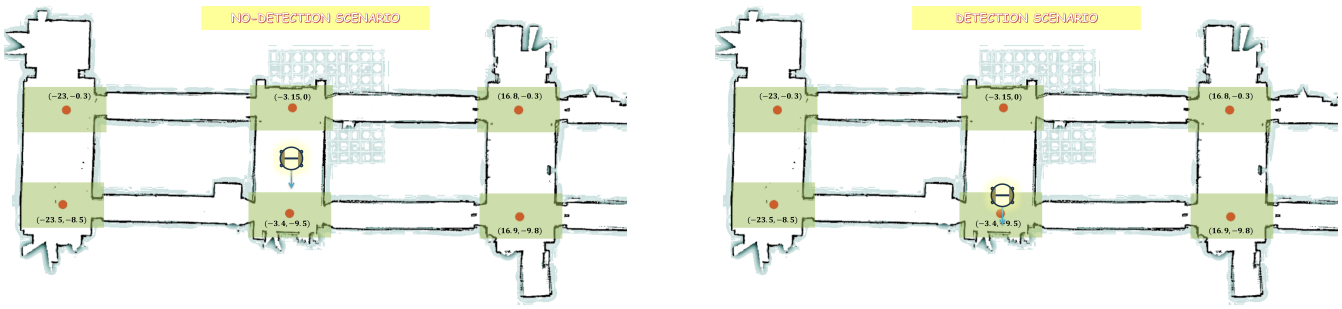


Fig. 6 – Flowchart illustrating the `navigation_manager` method in the Turtlebot4 navigation system.



(a) No-Detection Scenario

(b) Detection Scenario

Fig. 7 – Illustrative scenarios demonstrating Turtlebot4 unable to detect QR codes (a) and successfully detecting them (b).

QR Code Detection Logic The QR code detection logic in the navigation system of the robot is designed to allow it to dynamically respond to commands encoded in QR codes placed in its environment, assuming the Turtlebot4 is capable of acquiring a new command from the QR code only when it reaches the designated arrival junction. This **junction** is defined by a rectangular area centred on the waypoint, with dimensions of 7 meters in width and 5 meters in height, enabling the reception of QR code signals read from the topic `/cmd_qr`. Here's an explanation of how this logic is handled:

- **Detection Before Reaching the Waypoint:** When a new command is received before the robot reaches its intended waypoint, the `cancelTask()` method is invoked. This method cancels the current navigation task, ensuring that the robot halts its current path or action in favour of executing the new command.
- **Detection After Reaching Waypoint:** After reaching the waypoint, the Turtlebot4 continues to monitor its surroundings for new QR codes. The camera's resolution and range enable detection even after reaching the target waypoint. This capability allows the robot to receive and execute commands placed beyond the initial target, maintaining flexibility in navigating through dynamic environments. In scenarios where no new QR Code signal is received immediately upon reaching the waypoint, the Turtlebot4 initiates a rotational scan. It rotates incrementally, typically in 30-degree steps within the range of [–90, 90] degrees, to scan for QR codes that might have been missed initially.

Handling Unexpected Movements (Kidnapping) The Kidnapped Manager plays a pivotal role in handling critical scenarios to ensure robust and reliable operation. It manages the robot's state in three primary conditions:

- **Failed Command Reception and Rotation Attempts:** This scenario occurs when the robot fails to receive a command from the QR code after attempting to acquire it through rotation. The failure could stem from frames shifting during rotation or from a QR code being positioned inaccurately.
- **No New Waypoint Found:** If the system fails to identify a new waypoint for navigation, it indicates a critical failure in navigation planning. This typically results from the robot's incorrect orientation upon reaching the destination waypoint.
- **Robot Kidnapped Detection:** The system monitors the kidnapped status of the robot via a subscription to the `/kidnap_status` topic. It triggers the Kidnapped Manager upon detecting that the robot has been unexpectedly moved while navigating.

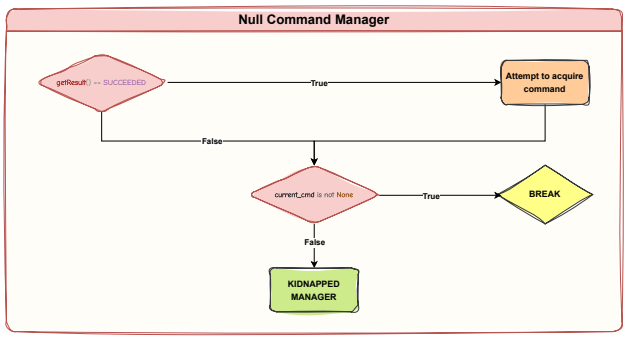


Fig. 8 – Flowchart illustrating the `null_command_manager` method.

Upon detecting a kidnapping event, two scenarios are considered:

- **Repositioning with RViz:** If the user specifies a new position for the robot using RViz, the robot adjusts to this new pose. This scenario allows the robot to potentially continue its navigation task from the updated position if it was kidnapped during navigation through landmarks [1].
- **Repositioning to the last known waypoint:** If no new position is specified in RViz, the robot assumes it has been repositioned to the last known waypoint. In this case, the robot's current command is set to `STRAIGHTON`. If the robot was kidnapped during navigation, the current task is cancelled, and navigation restarts from the new pose.

These strategies should collectively enhance the system's adaptability and resilience in dynamic environments, ensuring high fidelity in navigation performance.

Fig. 9 – Example of Turtlebot4 being kidnapped.

1 **AMCL** uses a Monte Carlo method to estimate the robot's pose based on sensor data. When the robot is kidnapped (unexpectedly moved), AMCL struggles to update the pose accurately due to the abrupt change in its perceived location relative to the map. This is because the sudden movement exceeds AMCL's expected range of normal motion, leading to delayed or incorrect pose updates through the `/amcl_pose` topic. In contrast, **RViz**'s `/initial_pose` topic allows manual specification of the robot's pose, providing a more reliable method to reset the robot's position after a kidnapping event.

4 | Test

This section provides a comprehensive evaluation of the project from both qualitative and quantitative perspectives.

4.1 | Qualitative Test

The **qualitative tests** focus on the quality and usability of the project. They aim to assess how well the system performs in terms of user experience and overall satisfaction.

QRScanner Qualitative Test This paragraph shows the testing procedure for the `QRScanner` implementation, responsible for decoding QR codes from images captured by either the Turtlebot4's camera or the PC's webcam. The versatility of this implementation allows for seamless integration and testing in different environments. In both scenarios, the node subscribes to a specific topic to receive images published either from the PC's webcam (`/camera_image`) or from the Turtlebot4's camera (`/oakd/rgb/preview/image_raw`). Upon successfully detecting and decoding QR codes from the received images, the node proceeds to publish any decoded commands to the `/cmd_qr` topic.

```
# Test using the PC's webcam
ros2 launch qr_commaner qr_launch.py on_turtlebot:=False
# Test using the Turtlebot4's camera
ros2 launch qr_commaner qr_launch.py on_turtlebot:=True
```

Listing 2 – These commands demonstrate how to initiate testing of the `QRScanner` implementation using both the Turtlebot4's camera and the PC's webcam.

Fig. 10 – Example of Turtlebot4 detecting the `GOBACK` command in an operational context.

Turtlebot4 Navigation Qualitative Test This paragraph provides a detailed guide on connecting to and operating the Turtlebot4, essential steps for testing the developed code effectively on the robot. Before testing the code on the Turtlebot4, establishing a connection is paramount. This process is streamlined through a dedicated shell script, which automates the necessary steps for connecting to the robot. After ensuring that the device is on the same network as the Turtlebot4, execute the following command in the terminal: `./turtlebot_init.sh`. Once connected, the Turtlebot4 can be maneuvered within the environment using the following commands executed in separate terminal tabs:

1. **Load the Map:** Launch the `localization.launch.py` file from the `turtlebot4_navigation` package, specifying the map file `diem\_map.yaml` located in the `src/map` directory.
2. **Run Navigation Stack:** Launch the `nav2.launch.py` file from the `turtlebot4_navigation` package to activate the navigation functionality of the Turtlebot4.
3. **Open RViz:** Launch the `view_robot.launch.py` file from the `turtlebot4_viz` package to set up the visualization environment for the Turtlebot4.
4. **Run Custom Navigation Implementation:** Launch the `nav1.launch.py` file from the `nav_pkg` package to execute the specific navigation implementation tailored for the `@DIEM map`.

```
# Load the map
ros2 launch turtlebot4_navigation localization.launch.py
↪  map:=src/map/diem_map.yaml
# Run navigation stack
ros2 launch turtlebot4_navigation nav2.launch.py
# Open RViz
ros2 launch turtlebot4_viz view_robot.launch.py
# Run our navigation implementation for @Diem map
ros2 launch nav_pkg nav1.launch.py
```

Listing 3 – These commands are crucial for initiating the testing process of the Turtlebot4 Navigation system.

After executing these commands, select the Turtlebot4's initial position and orientation in RViz. By default, the Turtlebot4 will commence movement in a straight direction, following the command `Commands.STRAIGHTON`.

4.2 | Quantitative Test

The **quantitative tests** involve objective measurements and data-driven assessments of the project's performance.

QRScanner Quantitative Test Before selecting our QR detector, we evaluated various detection methods available in the literature, including `OpenCV`'s QR detector, `Pyzbar`, and `QReader`. Our aim was to determine the most suitable detector for our application, considering the following factors:

- **Detection Distance:** We evaluated the detector's ability to detect QR codes at varying distances. `QReader` emerged as the top performer. Our tests indicated that the detector is capable of detecting QR codes even beyond 1 meter. However, it's essential to note that while detection may occur at greater distances, the accuracy of decoding decreases significantly. This suggests that while the detector can identify the presence of QR codes in the image, it may struggle to extract meaningful data accurately. Improving the detection accuracy at longer distances could involve enhancing the detector's robustness to image noise and implementing advanced image processing techniques to mitigate the effects of distance-related challenges.
- **Decoding Distance:** To assess the decoding accuracy of each detector, we conducted tests to measure the maximum distance from which QR codes could be decoded. Once again, our findings revealed that the `QReader` is the top performer compared to other detectors. Specifically, decoding accuracy remained consistently high at distances of up to approximately 1 meter. Within this range, QR codes were decoded accurately and reliably. However, beyond this threshold, which we defined as the decoding distance, we observed a noticeable decline in decoding accuracy. Several factors contribute to this decline, including image distortion, noise, and reduced image clarity at greater distances from the QR code. These findings underscore the importance of optimizing detector parameters to enhance performance at longer distances.

Method	Pyzbar	OpenCV	QReader
Max Rotation Degrees	17°	46°	79°

Table 1 – Comparison of the max rotation degree of each detector.

- **Rotation Tolerance:** We also compared the maximum rotation tolerance of each detector. As indicated in its GitHub repository [1], `QReader` exhibited the highest tolerance at 79°, surpassing both `Pyzbar` and `OpenCV`. This capability ensures that `QReader` can reliably detect and decode QR codes even when they are rotated to a considerable extent, enhancing its versatility and applicability in real-world scenarios.

Overall, our quantitative evaluation reaffirmed the superiority of `QReader` in terms of both detection and decoding accuracy. These findings solidify `QReader` as the optimal choice for our `QRScanner` implementation, ensuring reliable performance across various distances and orientations. However, we were not satisfied with the performance. This is because a reduced field of view could limit the TurtleBot4's ability to detect signals from QR codes effectively. To address this, we adjusted the camera parameters in RViz as detailed below. These modifications enabled us to achieve a detection and decoding range close to and slightly above 3 meters.

```
/oakd:
    camera:
        i_enable_imu: false
        i_enable_ir: false
        i_floodlight_brightness: 0
        i_laser_dot_brightness: 100
        i_nn_type: none
        i_pipeline_type: RGB
        i_usb_speed: SUPER_PLUS
    rgb:
        i_board_socket_id: 0
        i_fps: 15.0
        i_height: 1440
        i_interleaved: false
        i_max_q_size: 10
        i_preview_size: 1000
        i_enable_preview: true
        i_low_bandwidth: true
        i_keep_preview_aspect_ratio: true
        i_publish_topic: false
        i_resolution: '4K'
        i_width: 2560
```

Listing 4 – Our Turtlebot4's camera configuration parameters.

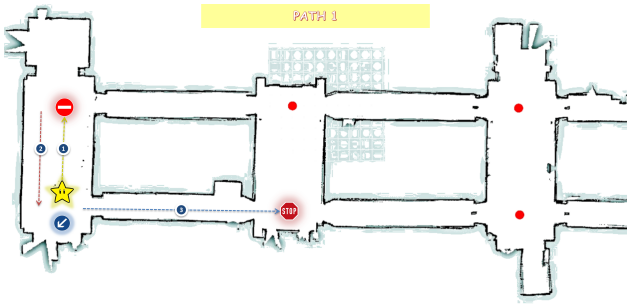
(a) Successfully detected and decoded QR code from a distance of approximately 3 meters.

(b) QR code detected but not successfully decoded from a distance of approximately 3.5 meters.

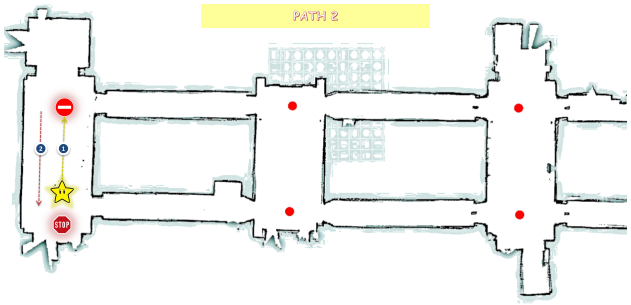
(c) Successfully detected and decoded a QR code from a distance of approximately 3 meters, albeit slightly rotated and positioned diagonally relative to the Turtlebot4.

Fig. 11 – Illustration of the QR code detection and decoding capabilities of the Turtlebot4's camera system under varying conditions.

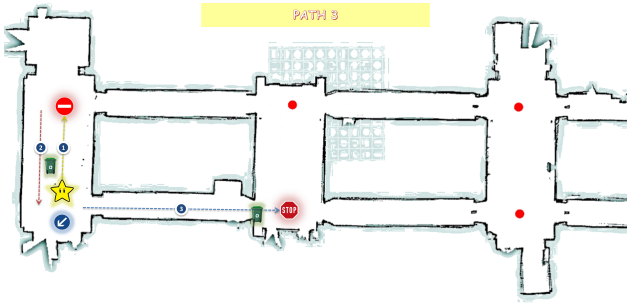
It is reasonable to deduce that due to fluctuations caused by the movement of the Turtlebot and the frame rate at which images are transmitted, the robot may not be able to detect QR code signals at the same distance when in motion.



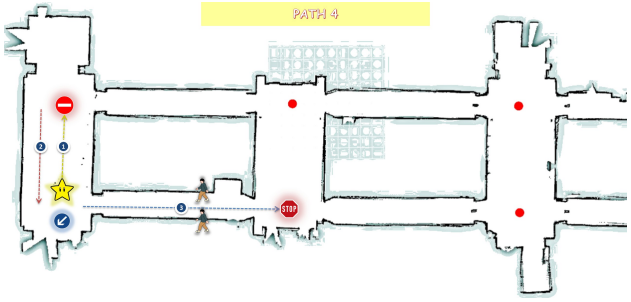
(a) `@Path1` used exclusively for Test 1, focusing on ideal navigation without obstacles.



(b) `@Path2` used for Tests 2 through 6, addressing QR code detection challenges and recovery strategies.



(c) `@Path3` designated for Test 7-8, involving navigation around static obstacles.



(d) `@Path4` employed for Test 9, incorporating dynamic obstacles to assess adaptive navigation.

Fig. 12 – Illustrations of the different paths used in the Turtlebot4 navigation tests. These paths ensure a thorough examination of the Turtlebot4's navigation performance under diverse conditions.

Turtlebot4 Navigation Quantitative Test This section provides a comprehensive overview of the main field tests conducted to assess the readiness and performance of our Turtlebot4 navigation algorithm. The following is a detailed summary of the implemented tests and their key outcomes:

- **Test 1: Ideal Simple Navigation.** This test scenario verifies the Turtlebot4's navigation accuracy in an environment free of both static and dynamic obstacles. As depicted in Fig. 12a, the robot follows the `@Path1`, encountering and processing several QR codes along the way. Specifically, the robot encounters a `GOBACK` signal, indicating it should reverse or retrace its steps. Following this, it comes across a `LEFT` signal, directing it to turn left at the specified point. Finally, the robot detects a `STOP` signal, instructing it to halt.
  - ✔ **Success Condition:** The test is successful if and only if the robot follows the entire `@Path1` without making any kidnap requests. The test was successful as show in the Video 1.
- **Test 2: Ideal Simple Navigation - QR Code Not Detected with Successful Rotation Recovery.** As depicted in Fig. 12b, this test involves the Turtlebot4 navigating along the `@Path2` without encountering any static or dynamic obstacles. Initially, the robot fails to detect the `GOBACK` signal due to its lateral displacement at the junction. To recover the QR code signal, the Turtlebot4 executes a rotational scan within a range of [-90, 90] degrees. This successful rotation allows the robot to reacquire the QR code signal without needing to initiate a complete repositioning procedure.
  - ✔ **Success Condition:** The test is successful if and only if the robot is capable of detecting the QR code with the rotational scanning logic, without making any kidnap requests. The test was successful as show in the Video 2.

- **Test 3: Ideal Simple Navigation - QR Code Not Detected with Robot Kidnapping and New Position Set in RVIZ.** This scenario tests the Turtlebot4's response when it fails to detect a QR code signal and rotational recovery proves ineffective. The robot is kidnapped, necessitating its repositioning using RVIZ. The new position in RVIZ is carefully selected to optimize the robot's ability to detect the QR code signal it failed to recognize previously.
  - ✔ **Success Condition:** The test is successful if and only if the robot is capable of detecting the QR code after being repositioned on the floor. The test was successful as show in the Video 3.
- **Test 4: Ideal Simple Navigation - QR Code Not Detected with Robot Kidnapping and New Position Not Set in RVIZ.** Similar to Test 3, the robot fails to detect the QR code and rotational recovery is unsuccessful. The robot requests kidnapping and, in this case, the new position is not specified in RVIZ. Once repositioned on the ground, the robot assumes it is at the last visited waypoint, oriented towards the destination waypoint.
  - ✔ **Success Condition:** The test is successful if and only if the robot is capable of detecting the QR code after being repositioned on the floor. The test was successful as show in the Video 4.
- **Test 5: Ideal Simple Navigation - Robot Kidnapped during Navigation with New Position Set in RVIZ.** During navigation towards the target waypoint, the robot is unexpectedly kidnapped for unspecified reasons. The robot is repositioned directly at the junction by specifying the exact new position using RVIZ. Once repositioned in the physical environment, the Turtlebot4 resumes execution of the pending navigation task.
  - ✔ **Success Condition:** The test is successful if and only if the robot executes `@Path2` correctly after being repositioned on the floor. The test was successful as show in the Video 5.

Fig. 13 – (Test 7) Navigating around multiple trash bins.

- **Test 6: Ideal Simple Navigation - Robot Kidnapped during Navigation with New Position Not Set in RVIZ.** In this scenario, the robot is kidnapped during navigation towards the target waypoint. Without a new position specified in RVIZ, once repositioned on the ground, the robot assumes it is at the last known waypoint, oriented towards the destination waypoint.
  - ✔ **Success Condition:** The test is successful if and only if the robot executes `@Path2` correctly after being repositioned on the floor. The test was successful as show in the Video 6.
- **Test 7-8: Real Simple Navigation - Presence of Static Objects.** As illustrated in Fig. 12c, this test involves the Turtlebot4 navigating along the path designated as `@Path3`, which replicates the navigation path used in the first test. However, this scenario introduces static obstacles, such as trash bins, which may obstruct the robot's path. The goal is to assess the robot's capability to manoeuvrer around these static objects while adhering to its intended trajectory.
  - ✔ **Success Condition:** The test is successful if and only if the robot executes `@Path3` correctly. The test was successful as show in the Video 7.

  A similar successful test, as shown in the Video 8, was conducted by navigating the Turtlebot4 around two ladders positioned along the map.

Fig. 14 – (Test 8) Navigating around two ladders.

- **Test 9: Real Simple Navigation - Presence of Dynamic Objects.** As illustrated in Fig. 12d, this test involves the Turtlebot4 navigating along the path designated as `@Path4`, which replicates the navigation path used in the first test. However, this scenario introduces dynamic obstacles, specifically people moving in the opposite direction, which may obstruct the robot's path. The goal is to assess the robot's capability to dynamically adjust its path to avoid collisions while still following the intended route.
  - ✔ **Success Condition:** The test is successful if and only if the robot executes `@Pat4` correctly. The test was successful as show in the Video 9.
- **Test 10-11: Real Complex Navigation - Presence of Static and Dynamic Objects.** This test scenario verifies the Turtlebot4's navigation accuracy in a challenging real-world environment that includes both static and dynamic obstacles. As depicted in Fig. 15, the robot follows the `@Path5`, encountering and processing several QR codes along the way. Specifically, the robot encounters a `GOBACK` signal, indicating it should reverse or retrace its steps. Following this, it comes across a `LEFT` signal, directing it to turn left at the specified point. Subsequently, it detects a `STRAIGHTON` signal, directing it to proceed straight ahead. Upon reaching the next waypoint, the robot encounters another `LEFT` signal. Finally, the robot detects a `STOP` signal, directing it to come to a halt, thereby concluding the navigation task.
  - ✔ **Success Condition:** The test is successful if and only if the robot follows the entire `@Path5` without making any kidnap requests. The test was quite successful as show in the Video 10.

  A similar test, as shown in the Video 11, was conducted by navigating the Turtlebot4 around a different path.

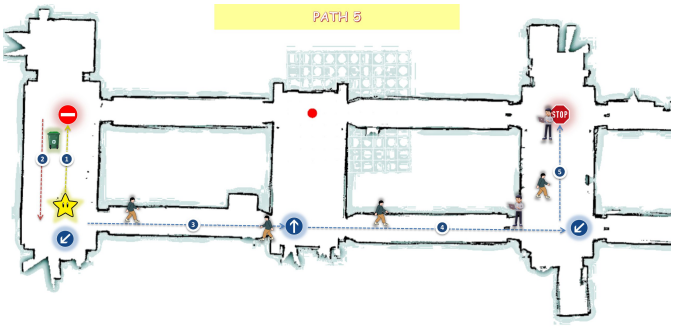


Fig. 15 – `@Path5` tests the Turtlebot4's ability to navigate a complex route amidst static and dynamic obstacles.

References

[1] Eric Canas. QReader: Robust and Straight-Forward solution for reading difficult and tricky QR codes within images in Python (powered by YOLOv8). https://github.com/Eric-Canas/QReader, 2024. Repository GitHub.